On API usability: An analysis and an evaluation tool

Andre Doucette

Exploratory Topic

CMPT 816 - Software Engineering

University of Saskatchewan

11 December 2008

Introduction

Software systems are becoming increasingly complex. As code bases increase in size, there is a need to modularize the code into functional parts, which allows programmers to focus on only a small subset of the entire functionality at one time. This also means that no one programmer needs to know everything about every part of the code base.

One of the most common ways to achieve this goal is by providing an application programming interface, or API. An API is exactly what it sounds like; it is an interface between a large code base and a programmer. This limits the programmer's access to the code by requiring access to the code base only through this defined (and constrained) interface.

The problem is that programmers can see only the documentation and the interface itself. This means the only connection between the functionality the code provides and the programmer is the interface set up between them. Like all interfaces, it becomes difficult to convince others that this new interface is worth using if it is difficult to use, impossible to understand, or even if the programmers just don't like it. This is the motivation for this project. APIs are a very useful organizational tool, but a tool is only useful if people use it. Creating an API which programmers enjoy to use but also allows the programmers to be productive while doing so is an important aspect of API design. These, among other features, are what defines API usability.

There is little research into API usability, but the current literature often measures API usability qualitatively by conducting usability testing. These tests are expensive and time-consuming. For this reason, I propose creating an automated API evaluation system, which rates APIs on a series of metrics that were created from informal (and formal) discussions from various sources. This system is unique, in that it evaluates an API quantitatively, in contrast to the usual qualitative evaluations currently employed.

A first step in creating an evaluation system was developed, and a few analyses were conducted on the results. It was interesting to see the amount of information that can be collected by an automated tool, but future work should investigate the proposed metrics in more detail, as well as other metrics (possibly including a syntactic analysis of the API). It is hoped that this project motivates the creation of a set of quantitative heuristics which can automatically rate an APIs usability.

Usability

Software quality

One could argue that CMPT 816 is a course about software quality, with some advice that will allow programmers to avoid some common pitfalls. Quality is one of those words that is used almost ubiquitously, but with each usage, it has a different meaning. In the context of software, there are some standard definitions for the parts that comprise it, including an ISO standard [17] and various corporate definitions. Although each definition may use a different set of words to describe what quality means, there are usually many commonalities between them. These often include functionality, performance, and ease-of-use.

Functionality is loosely described as whether the users of the software are capable of accomplishing the tasks they set out to do. Performance deals with how efficiently the software is able to perform the tasks the user wishes to accomplish. Lastly, ease-of-use, which is possibly one of the more contentious aspects of software quality, is a subjective measure from the users which represents how easily the task was accomplished, how much the software helped them along the way, and how many errors were performed, among other aspects. Ease-of-use generally falls under the umbrella of usability.

Usability overview

The ISO 9241 standard defines usability as “the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use” [19]. This definition states that the extent to which a product is usable depends on both the user and the context in which the user is attempting to accomplish a specific task. Given a different task or a different context, the same user may have a different idea of what would make the product usable. There is one problem with this definition: the words effectiveness, efficiency, and satisfaction often have different meanings for different users. Generally, one can think of these as being able to accomplish your goal (effectiveness), accomplishing your goal quickly and with few errors (efficiency), and feeling content with your performance (satisfaction).

There are five generally accepted subparts to system usability [7]:

1) Learnability - Should be easy to learn how to accomplish tasks in the system, so users can quickly begin working
2) Efficiency - Once users have learned how to use the system, they should be highly productive using it (at least more productive then without the system)
3) Memorability - Should be easy to remember for casual users, so they don’t have to relearn the entire system every time they return to do more work
4) Errors - Should prevent the users from making errors, should fail gracefully, and should be able to recover easily from errors
5) Satisfaction - Users should have positive subjective feelings about using the system (i.e., do they like using it)

Combining the above criterion helps decide whether users will find a system usable or not; however, several issues are raised. The most important is “How do you measure these criteria?” Other concerns include the motivation for doing this (or “why do I care”), cost of doing the analysis, cost of adhering to the criteria, and the benefits to the users of the system. These topics are discussed throughout this report.

Why is usability important?

At its core, usability is important because it increases the users’ satisfaction with the product. The subjective feelings of users are important due to the word-of-mouth feedback the users will pass along. If this feedback is positive, it will hopefully encourage other users to adopt your product.

Other than making your users happy, usability has a signifiant impact on business costs. For example, let's assume that your product is a word processor. Your customers will buy your word processor because they hope it will increase their employees' efficiency. If your product is very efficient for expert users (efficiency), this is good for those expert users, but if this product has a huge learning curve, much time will be wasted when novice users try to learn how to use your product (learnability). This learning time is expensive for organizations, who want their employees to be as efficient and productive with the new software as quickly as possible.

This simple example shows that the aspects of usability are often tradeoffs. It is very difficult to completely fulfill all aspects of usability, but some aspects are complementary. For example, a user who feels that they were able to learn the system quickly (good learnability) will likely have good subjective feelings about the software (satisfaction).

Application Programming Interfaces (APIs)

Definition

An application programming interface (API) is the link between a programmer and the (hidden) code base behind it. The abstraction that APIs provide is very common in computer science. One programmer, who is an expert in one particular area, can write a piece of code that can be used by other programmers without those programmers having to be experts in that area. Systems that provide this abstraction have numerous names, including: libraries, frameworks, development kits, toolkits, and APIs [12]. Although each of these terms has a specific meaning in whatever context it is used, the common factor in all of them is that there is a code base (usually written by someone else) that a programmer wants to use in their own code. Instead of copying and pasting the source code (which may not even be provided), an interface is developed that links the supplied code and the programmer's new code. This interface is what I will be referring to as an API, which includes any way that a programmer can access this code base.

Over the years, many programmers have written huge amounts of code and made the code available to the public, either for free or for a cost. These code bases cover varying aspects of functionality, from image processing, to linear algebra, to networking, and even operating systems.

These code bases are often presented in one of two ways: as classes and methods (for object oriented APIs), or as a series of functions (for non-object oriented APIs). Since the new programmer can only access the code base through the interface, the new programmer only has access to the names of the classes, methods, and functions, and their parameters. They do not have direct access to the code base itself, but can use the functionality provided by using the interface.

API usability

As discussed in the previous section, systems can be evaluated on their usability. Although there is a large amount of literature on usability, there is very little dealing with API usability. I wondered why this was the case, which motivated the work in this project. Programmers are users too; they are using APIs everyday, usually through a high-level programming language. As programmers' experience varies, each programmer has a different idea of how they will approach a problem. This is called the programmers mental model. In general, programmers rarely know only one programming language and paradigm, and so are accustomed to having to change their way of thinking to adapt to the problem they are currently trying to solve. This is not necessarily a simple task for a programmer. Quite the contrary; many programmers find it very difficult to begin programming in a new language or using a new API. This project focuses on the learnability and memorability issues relating to APIs. It is hoped that through this, new APIs can be designed in such a way as they are easier to learn and easier to use for programmers.

Object-oriented APIs expose classes, methods, functions, and fields to the programmers. Since these (and documentation) are the only parts exposed to the programmers, these define the barrier between the programmer and the functionality provided in the code base. It is assumed in this project that API designers will expose parts of the API to the programmers by using public classes and methods; however, this assumption may be invalid if the API designers are not consistent with their use of public classes and methods.

So what makes an API usable? Since programmers can only see the exposed parts of the API, the naming of the classes, methods, and fields are the most important factor in API usability. If the names are chosen appropriately to match the programmer's mental model, then the programmer is more likely to be able to guess what the name of the class or method is they are looking for. For example, if a programmer is trying to add an object to a stack, the programmer may try looking for the method *push*. If the method is actually called *insert*, then the programmer is unlikely to guess it and may need to resort to scanning all the available methods to find a method name that seems to mean "add this object to this stack".

Other parts of an API which should match the programmer's mental model include the structure of the classes (which classes are available and their inheritance hierarchy), as well as the functionality of those classes (the methods). A common idiom in the programming world is "Simple tasks should be easy to accomplish, and hard tasks should be possible"; however, the word "easy" is very subjective: One programmer may find the API matches their mental model, whereas another programmer may find that it is completely the opposite of what they are imagining. This duality makes it extremely difficult to design a truly usable API, and so there will always be a set of the users for which your choice of names and functionality will not match what they are looking for. Bloch [2] offers a good solution: "You can't please everyone so aim to displease everyone equally."

Documentation and examples

Most of the current literature on the topic of API usability gives some general guidelines that should be followed while developing an API. Many of these [1,2,3,9] state that one of the best methods of ensuring your API will be usable is to code examples against the API from the first drafts. These code examples are the typical use-cases, or tasks, that the API is meant to support. By designing your API with a specific set of tasks in mind, you can ensure that you include the parts that are needed, and remove the parts that are not needed. This obeys the general “less is more” idiom stated by many API usability experts [1,3,4].

These examples are useful, because they can be provided to the user to illustrate how your API was meant to be used. Stylos [14] and others found that programmers were more likely to use example code than read documentation when trying to learn a new API. This is an important consideration, because documentation (if available) is often the focus of API designers, not the example code. By creating these functional chunks illustrating how the API should be used, and providing them to the user, you are providing a mechanism for programmers to learn how to use your API.

The role of tools in API usability

If a programmer was required to memorize all the classes, methods, and fields provided by the API, it would likely be extremely difficult, time-consuming and error-prone. This is where documentation becomes useful, providing information to programmers on parts of interest. With the newest set of integrated development environments (IDEs), we can supplement the documentation with features such as code completion and Microsoft’s IntelliSense [17]. These features expose an easier mechanism for exploration [1,2], and provide recognition of class and methods names versus the recall of these names. Recognition versus recall is one of Nielson’s 10 usability heuristics [12,13], which are one of the most commonly accepted set of usability heuristics.

What this means is that tools can reduce the need for memorability from the user, but it is possible that they may increase the learnability as well. According to [14], programmers use code examples much more than regular documentation. It was also found that programmers, at least in an object oriented environment, are more likely to instantiate an object and use an IDE’s features to display the methods and fields that are available to them. This is an interesting method to learn an API; unfortunately, these methods are displayed in alphabetical order, and are not grouped by the types of functionalities these methods provide. Also, if each class has hundreds of methods, the utility of this tool is suddenly reduced. With a large number of methods, the programmer must begin typing the name of the method to narrow down the results; however, this increases the need for programmers to memorize the API.

Other researchers have begun to investigate the user of tools in API usability. The Strathcona Example Recommendation Tool [8] is a good example of this investigation. This automated tool will search a large database of example code to provide the programmer with relevant code illustrating how others have used this part of the API. Integrating the use of this tool into a large development project is a great idea, but requires a large amount of example code to have already been developed before it is of any use; however, if the API designer has been coding against their API from the beginning, then these examples can be provided to the programmer and may help guide them to use the API correctly (i.e., in the way it was meant to be used).

Another useful tool similar to Strathcona is Prospector [11]. Prospector is a query-based tool, which allows programmers to use a jungloid (a unary query) that returns a code snippet that illustrates recommended usage of the class or method in question. These code snippets are mined from large existing code bases, and thus has similar issues to Strathcona.

Metrics

Introduction

The focus of this project is on the automatic evaluation of API usability against a set of metrics. These metrics were developed by reviewing the current literature, finding the common suggestions given, and seeing if these could be converted into a value that could be calculated automatically. For many of the more robust heuristics, it was not initially clear how a computerized solution would evaluate them. For example, “Keep APIs free of implementation details” or “You can’t please everyone so aim to displease everyone equally” from Joshua Bloch’s excellent talk “How to Design a Good API and Why it Matters” [2] are both subjective ideas, and would be very difficult to code a recognizer for these metrics. Other guidelines, however, are especially appropriate for an automated system. For example, “Avoid long parameter lists” [1,2] and provide a default constructor [16] can, in some way, be evaluated by an automated system.

Still other guidelines seem to be a good match for automated evaluation, but would require implementation of some semantical analyzer to associate meaning to the names assigned to fields, methods, and classes in the API. For example, if an API uses the method name “add” for some collections and the name “put” in other collections, this is inconsistent naming (as is present in the Java API). To a programmer, you “push” an element on a stack but “add” an element to a set, which is where the semantical meaning of the names would require some interesting processing. From another naming perspective, “Use consistent parameter ordering across methods” [1,2] is an example of a guideline which could be used in an automatic system. If, for example, a graphical user interface (GUI) API, such as Swing, has methods that take in x, y, width, height parameters in some methods, and take width, height, x, y parameters in other methods, the programmer will undoubtably become confused about which methods require which parameter ordering. These are possibly the most interesting aspects of API usability, but are unfortunately beyond the scope of this project. This implies that it is possible that the best evaluation mechanisms are beyond the scope of this project.

Hypothesis

It is the public classes, methods, and fields that determine the usability of an API, and an objective, automated analysis of these parts of an API will give valuable feedback on the usability of an API.

This is potentially a contentious issue, since there are many other factors that play a role in the usability of an API. For example, the previous experience of the programmer may play a role in their understanding of the API’s model (i.e., it matches their mental model, or conversely, is different from what they are expecting).

The metrics

1) Number of classes
2) Number of non-exception classes
3) Depth of inheritance hierarchy for each class
4) Number of constructors for each class
5) Number of methods each class contains, and an average over all classes

6) Number of overloaded methods, and are they distinguished by the number of parameters, the type of parameters, or the ordering of the parameters?
7) Number of parameters in each method
8) Number of public member fields in each class, and whether they are capitalized (as is the convention for constants in Java). Are these fields final?
9) Number of static methods and static final fields
10) How many static methods return an API defined object
11) Number of primitive type parameters and complex parameters (i.e., class objects) in methods and are these common Java objects or API-defined objects
12) Do the "set" methods return void (and presumably throw exceptions), boolean, or int (or another flag, like char)? Is it consistent across the API?

Discussion of metrics

It is not known which aspects of an API would provide a good metric to give valuable insight into the usability of an API, so I collected as much data as possible to investigate various aspects of API usability. Many of the metrics listed were created when considering guidelines and heuristics given in the literature, including Joshua Bloch's *Effective Java* book [1], a talk he gave on API usability [2], various papers from Jeffrey Stylos [14,15,16], and blog discussions from Microsoft and other sources [3,4]. The heuristics and guidelines provide hypotheses of what makes a good API, and these have been modified so that they can be calculated automatically by the evaluation tool. For example, it is assumed that fewer method parameters (7) and fewer overloaded methods with only the ordering of parameters changed (6) will allow the programmer to remember which class and method to use. Also, a smaller number of classes (1) and a shorter inheritance tree (3) should reduce the memory complexity required of the programmer.

Combining (4), (9), and (10) should indicate whether this class uses a Factory design pattern or not (i.e., if there are no public constructors, some static methods, and at least one of these static methods returns an API-defined object, this is probably a Factory class). We know if an object is a standard Java object or an API-defined object by checking the package name (i.e., if the root package is Java, it is a standard object). This pattern could indicate the use of the singleton pattern as well.

Bloch [2] states that methods should not return a value that requires additional processing. A method should return null or an empty collection instead of returning a boolean or an error flag. Interestingly, some "set" methods return a boolean to indicate whether the set was successful or not; others return void and throw an exception when the parameter is not valid; still others return some type of flag (for example, -1 if not successful). This inconsistency across multiple APIs is unfortunate, but should be consistent within one API and could indicate a problem in the API design if not consistent (12).

My tool

Introduction

Although the focus of this project is the investigation of API usability issues, a large portion of the work went into developing the API evaluation software, which outputs statistics on Java APIs according to the metrics described above. A few issues should be discussed before discussing the preliminary results.

Motivation for the evaluation tool

Previous work in API usability focused on heuristics and guidelines which arise during the development of the API. Many focus on qualitative results from user studies, which are expensive and time consuming. Others try to describe guidelines which should be adhered to while developing the API.

I have decided to take a different approach for this project: this evaluation tool reads in a Java jar file, and walks the classes in the file to extract the information required to evaluate the metrics for this API. This approach is particularly interesting for two reasons.

First, Bloch recommends coding against your API from the first draft of the API, even the first paper version. Since this tool is only interested in the structure of the classes, their methods, and the fields exposed to the programmer, you do not need to implement any of the functionality of the API before you can run the evaluation tests on it. Simply adding in stubs for the methods will not affect the usability measures, which means you can continually test changes to your API design to see how the changes will modify the usability metrics of your API.

Second, this is an automated system, which saves time and usability expertise. Simply load the evaluation software, plug in your API, and within seconds you will have a large amount of statistics about your API design.

Design choices

The usability evaluation tool was written in Java, and makes extensive use of the Java Reflection API to gather the information required for the evaluation. Alternative designs were explored, including converting source code to XML, then processing the XML separately with a language like XSLT. This proposed solution would have been portable to other languages, and would have allowed additional functionalities to be introduced (see Limitations section for details).

Regardless, the Java Reflection API was chosen for several reasons.

First, there are many Java APIs freely available online, and are downloadable as jar files. In other languages, it is unlikely that there would be as many APIs available for download that would provide the source code (to use the XML solution above, the original source code would be required).

Second, Java jar files are already represented in an object model (i.e., Java's object model). Other potential solutions, such as the XML solution proposed above would require processing the Java files to convert into another object model, which then gets converted again into a third model by the XSLT processing tool. Since Java already represents this information in an easily accessible model, the reflection API allowed the quick and efficient implementation of the evaluation tool.

What can my tool do?

For complete details on implementation, please see the included source code. The source is split into two source files: ClassCrawler.java is a convenience class used to parse the jar file provided as a command line parameter, and MetricsDriver.java calculates the metrics from the jar file provided. ClassCrawler is based mostly on code found online. The MetricsDriver class was written completely from scratch. It should be noted that the API designed was created in a very modular way, which allows multiple pieces to be linked together to get new functionality. In this way, new metrics can easily be implemented by simply choosing a different set of method calls. This implementation is easily modifiable, making it ideal as a built-in tool, since it simply calls various methods and outputs the results to a file. When new ideas surface, one can simply implement new methods and include them in the analysis.

Limitations

Jar files contain only the binaries and Javadoc (if available). It seems that method parameter names are not immediately available through the Java Reflection API. This seems strange, as modern IDEs (such as Eclipse) do provide you with the parameter names through their code complete features. This is disappointing, as some of the more interesting usability issues could have been investigated: for example, consistent parameter ordering. Even given this information, it would still have been difficult to implement because consistent parameter naming would need to be checked first (the ordering will not be the same if the naming is not the same). As discussed previously, this would require semantical analysis of the naming conventions, which is beyond the scope of this project.

The choice of using the Java language has a clear disadvantage. If the XML approach had been taken, this tool could be used for a variety of different languages. On the other hand, using the reflection capabilities of a language is an approach which could be adopted for other languages: only the syntax of how to accomplish the tasks would change, not the methodologies.

Another interesting point of choosing Java Reflection over XML is the loss of comments in the code (or, in the case of Java, the external JavaDocs). Assuming the writers of the API were conscientious programmers, their code will be marked up with comments, which could be a very useful tool for new programmers trying to learn how to use the API. Again, this may prove useful in usability measures, but would require additional semantical analysis (simply having comments is not enough; they should be complete and informative).

The most important limitation with using an automatic tool that inspects only the classes', methods', and fields' declarations is that documentation and examples are not included in the evaluation. Almost every discussion of API usability includes a section stating that documentation is very important for programmers trying to learn a new API. [8,15] states that programmers are more likely to use example code instead of pure documentation when trying to learn a new API. Had the XML solution been implemented, it may have been possible that the examples and documentation could have been included in the evaluation, but this would have been very involved and well beyond the scope of this project.

Lastly, it is possible that calculating metrics on an API may not be the best way to get a measure of API usability. Others have had success with usability testing [3,4], and have shown that approach works. Combining the automatic metrics analysis with user testing would be a good way to validate whether this is a good method to evaluate an API or not.

Preliminary results

It should be noted that I did run my own API through the metrics, and the results were interesting, to say the least. It seems that although the metrics may allow one to observe some type of API complexity and usability, the current implementation was thrown together to get the functionality without regard to the glaring breaking of many object-oriented idioms. The choice of using Java is not because Java and OOP are my primary programming tools, but because of the Java reflection API.

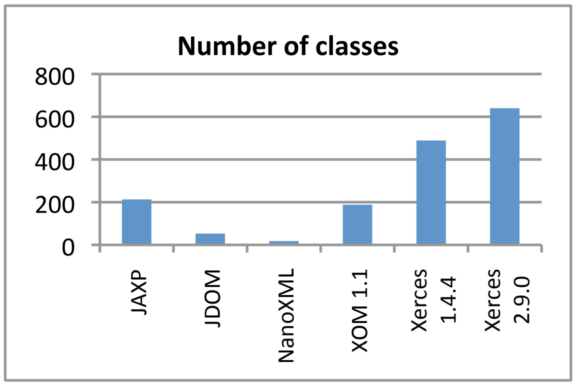
The evaluation tool tracks a series of different stats about the API, and can be easily changed to output different information in various formats. For an initial investigation, I have gathered a series of the most common Java XML APIs: JAXP, JDOM, NanoXML, XOM, and two versions of Xerces. In a second investigation, I collected the seven versions of Jakarta's Common Lang API, and ran each version through my program. Lastly, a quick survey of the Java API was completed, but due to the size of the Java API, it is difficult to draw any conclusions. These investigations were chosen for specific reasons.

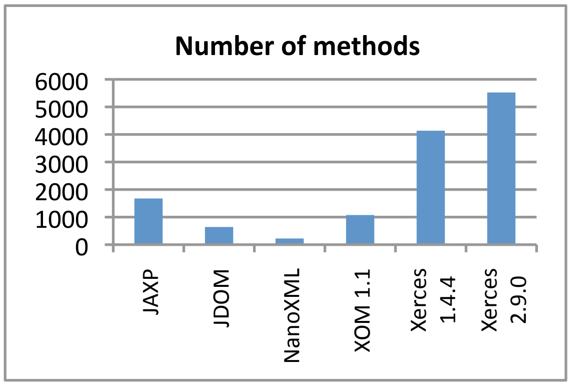
Comparing one API to another API may not be fair if the two APIs are designed to accomplish different tasks. As mentioned in the initial discussion, the usability of an API depends on the context and task, and is a subjective measure given to an API by the programmer currently using it. As such, it is likely that two programmers will rate an API differently. API usability is also context-sensitive; a programmer will expect different features of an API depending on the task they are trying to achieve. Thus, I have chosen one task domain (XML processing) and chose a series of APIs that are used within that domain. Granted, each of the libraries may be used by a different set of users. NanoXML is very lightweight, but Xerces is fully standard-compliant and extremely robust; however, the task that they accomplish are similar.

Another interesting approach is to investigate how an API changed over time, and see if the metrics could give any insight into how usable it was or has become. Presumably, an API will be updated over time, so most APIs have many different versions. Java style strongly discourages breaking backwards compatibility, meaning that upgrading to a new version of the API should not require any rework of the code base developed for the previous version. Without looking at the specific method and class names, as well as the documentation, it is difficult to track these types of changes but general trends of the API can be investigated. Also, it is assumed that as an API updates, it will either support more functionality (by providing more classes or methods), or will update the underlying implementation, making the implementation faster and more efficient. As mentioned in previous sections, this tool does not look at the implementation of the API's methods, simply at the interface provided to the programmers. So, plotting the metrics of an API over time should give us an indication of how the API has evolved over time. The Common's Lang API has gone through many iterations, and has grown to be quite large, so it was a good API to investigate.

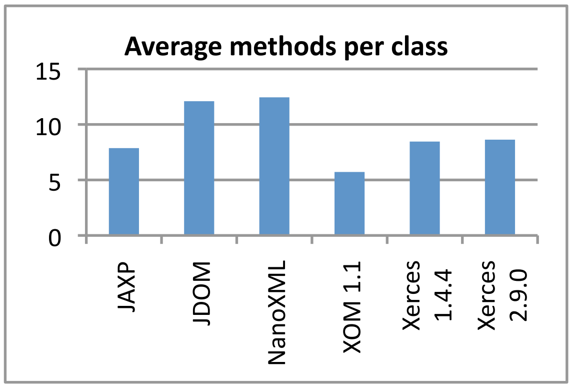
It is worth noting here that only a few of the metrics are described in the analysis, simply due to the overwhelming amount of data that can be gathered from the APIs. The evaluation tool gathers more information than is presented here, but a short discussion of some of the metrics is included to motivate future work in the area of automated API usability evaluation.

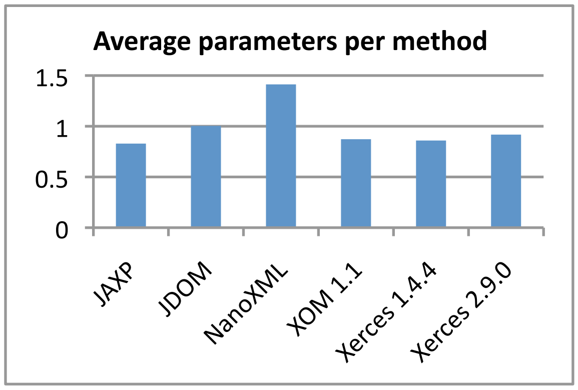
Part I - Six XML APIs

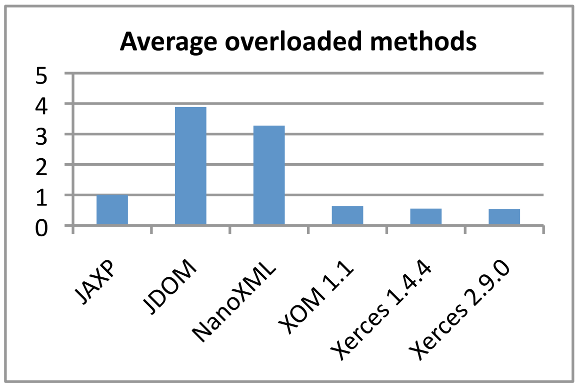




For this preliminary investigation, the six XML libraries were profiled and the results graphed above. Here, we look at the number of classes and total number of methods in API. Taken together, these can be thought of as representing the cognitive load the programmers must bear while coding for these APIs. As mentioned in the previous discussion, some of these APIs are meant to be lightweight, and the results clearly show this to be the case. It is worth noting that the difference in "weight" is rather significant: NanoXML has only 8 classes, JAXP has 213 classes, and the newest version of Xerces has 640 classes. The APIs with fewer classes also have fewer methods. This significant difference is interesting because all of these APIs are used to accomplish similar goals, which is parsing XML data and outputting XML data.





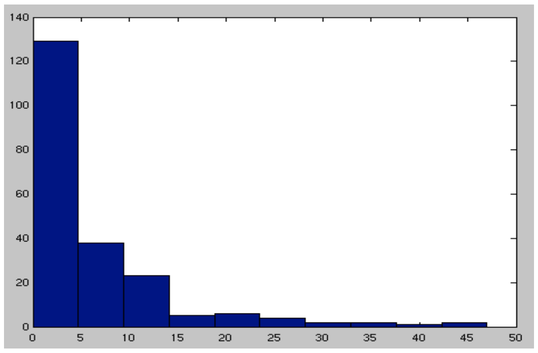


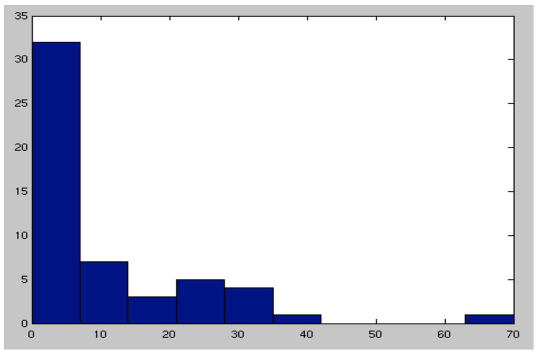
Here is when the discussion gets interesting. These three graphs show the average number of methods per class, average number of parameters per method, and the average number of overloaded methods for each class. Together, these represent how complex each class is (the number of methods in each class), as well as how complex these methods are (the number of parameters per method call and the number of times they are overloaded). We can observe that although the NanoXML and JDOM APIs contain significantly fewer classes and methods as compared to the Xerces and XOM APIs, thus making the overall cognitive load smaller, the complexity of those methods is larger: they take more parameters in each method call, and are overloaded more. Bloch states that an API designer should avoid long parameter lists, and that overloading should be done with care.

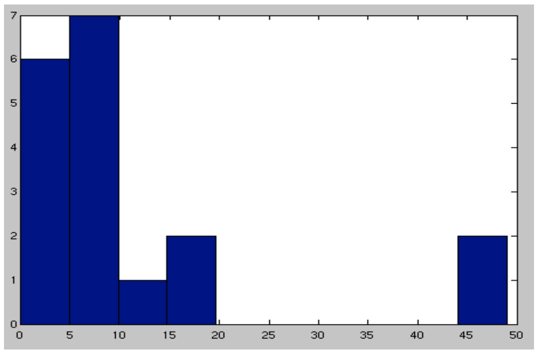
Long parameter lists increase the cognitive load that programmer needs to use it [2], similar to the amount of classes in an API. This also means that a programmer must spend extra time setting up the parameters before they are able to call the method, which was found to be counterproductive [15].

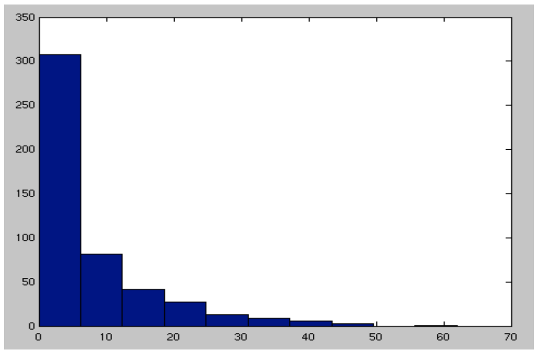
Method overloading can be a great convenience for a programmer, because they can reuse the same name but can allow for different input parameters; however, overloading must be done with care. There is a potential that users may confuse one method for another if the parameter lists are very similar but the functionality is different (why would you bother to overload a method if the functionality was the same?). This is often not an issue when the two method's side effects are the same (for example, a print method expecting a double when you pass it an integer), but here's an example where this may be an issue.

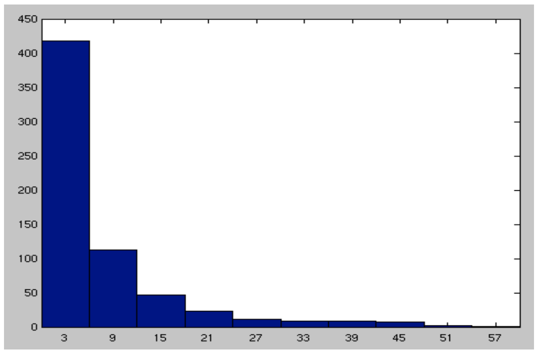
Assume a method that has been overloaded: setPosition(int x, int y) and setPosition(double x, double y). The first method sets the position to the absolute x and y values on the screen, whereas the second method sets the position on the same screen, but in normalized coordinates (say as a floating point number between 0 and 1). This is a potential hazard if the programmer doesn't realize there is this second method, and calls the method with setPosition(30.5, 10.2) assuming the method will simply truncate the values passed in. This can be considered a convoluted example, but as the size of the parameter lists grow, it is very easy to introduce these tricky situations.

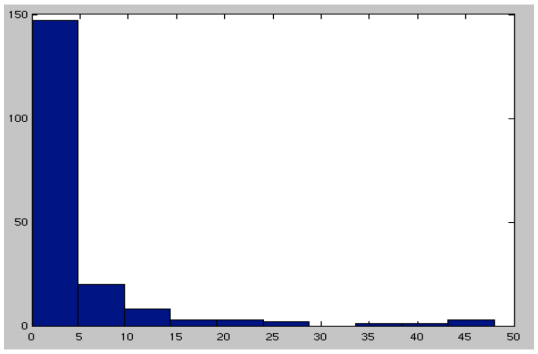

JAXP


JDOM


NanoXML

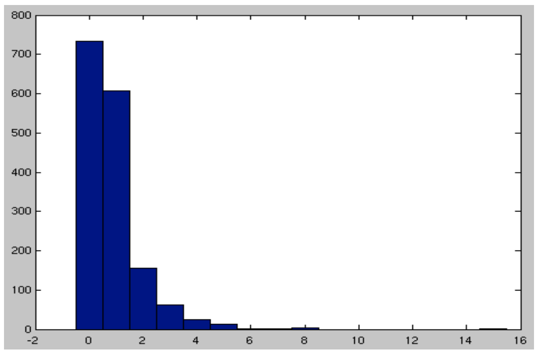

Xerces 1.4

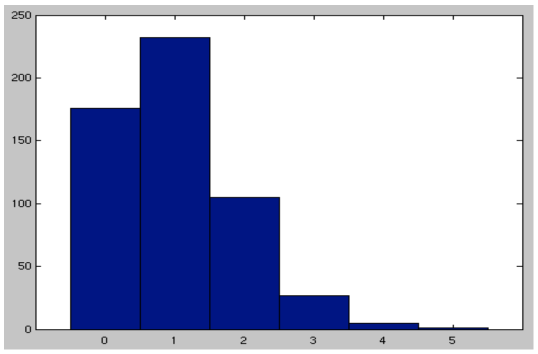

Xerces 2.9

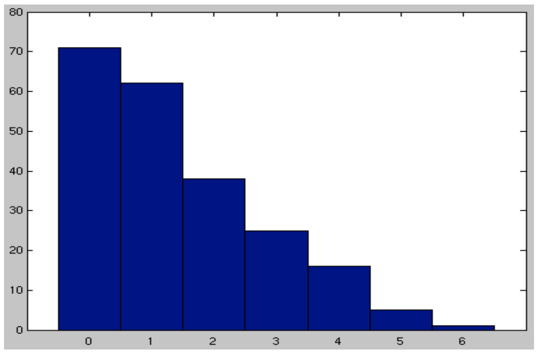

XOM

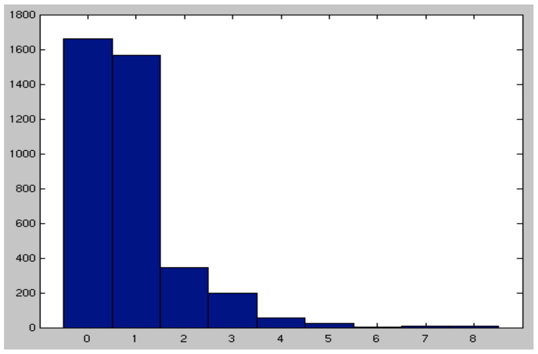
These histograms show the number of classes (y-axis) which have a certain number of methods (x-axis). Note that JDOM has one class which has 244 methods (which on its own may demonstrate the need to split a class into smaller classes), but this class was removed from the analysis as an outlier to make the graph easier to read.

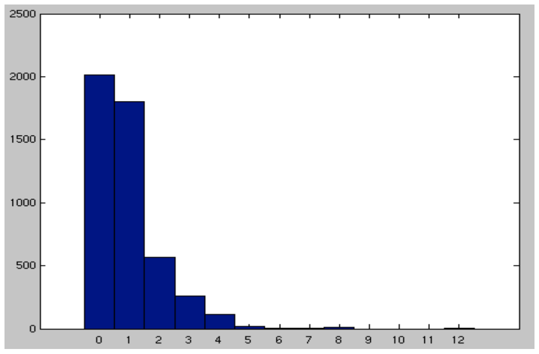
The histograms above serve a dual purpose. First, they help to illustrate that there are various types of information one can gather from using an automated analysis tool of APIs. Second, they show that these APIs have a trend towards making smaller classes, which is to say, classes that have fewer methods. These graphs, when combined with the graphs already discussed, elaborates on the story. It was previously stated that NanoXML had fewer classes, but that each of those classes were more complex than the classes of the other APIs; however, it appears that NanoXML's classes are no more complex than classes from other APIs. Specifically, all the APIs have more smaller classes than larger classes, somewhere around 20 methods. JDOM, however, has many of its classes in the 20-40 method range, which is more than most of the other APIs. I have not found any guideline or heuristic in the literature to indicate what is the optimal number of methods for a class, but this type of analysis over many classes and APIs could give interesting insight into what an appropriate number may be.

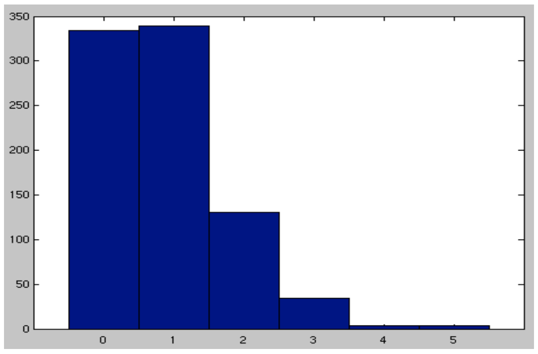

JAXP


JDOM


NanoXML

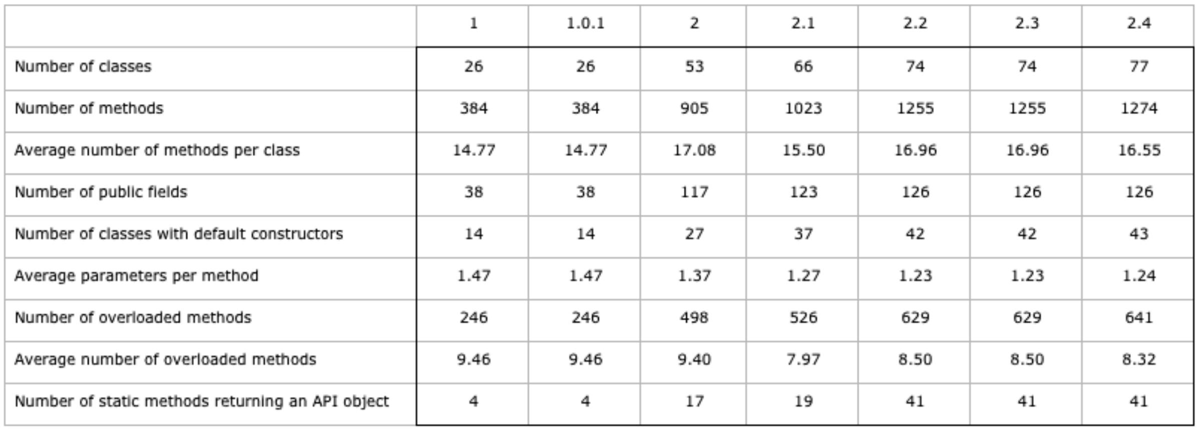

Xerces 1.4


Xerces 2.9


XOM

Here, we investigate the number of parameters per method for each of the APIs. The histograms plot the number of methods that fall into each of the ranges. Again, we can observe a general trend where methods should have 4 or fewer parameters per method. JAXP and Xerces rank the worst here, with methods with 15, 8, and 12 parameters, respectively; however, the number of methods in these categories is very small. Bloch [1] recommends that having 3 of fewer parameters per method is optimal, which is indirectly supported by Stylos [16].

Part II - Jakarta Common Lang over time



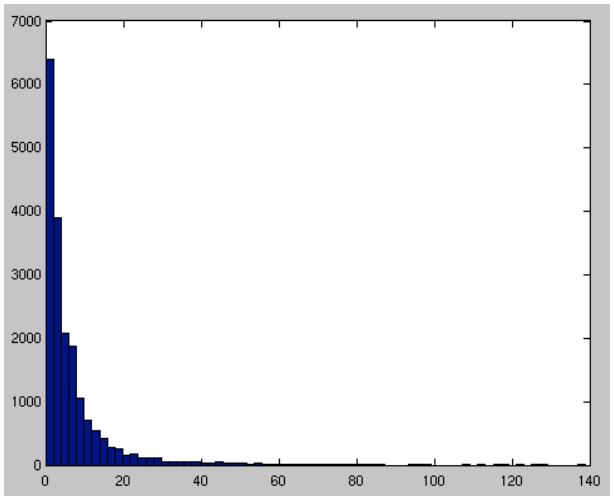
	1	1.0.1	2	2.1	2.2	2.3	2.4
Number of classes	26	26	53	66	74	74	77
Number of methods	384	384	905	1023	1255	1255	1274
Average number of methods per class	14.77	14.77	17.08	15.50	16.96	16.96	16.55
Number of public fields	38	38	117	123	126	126	126
Number of classes with default constructors	14	14	27	37	42	42	43
Average parameters per method	1.47	1.47	1.37	1.27	1.23	1.23	1.24
Number of overloaded methods	246	246	498	526	629	629	641
Average number of overloaded methods	9.46	9.46	9.40	7.97	8.50	8.50	8.32
Number of static methods returning an API object	4	4	17	19	41	41	41

The table above shows the progression of the Common Lang API from the initial 1.0 version up to the current 2.4 version. General trends can be quickly observed, which indicate the API is getting progressively more complex. For example, the number of classes and the number of methods almost tripled. Also, there was a significant change from the 1.x versions to the 2.x versions, but no changes are observed in the above metrics between version 1.0 and version 1.0.1. This is most likely because no new functionality was introduced, and the latter release had some bug fixes or performance improvements.

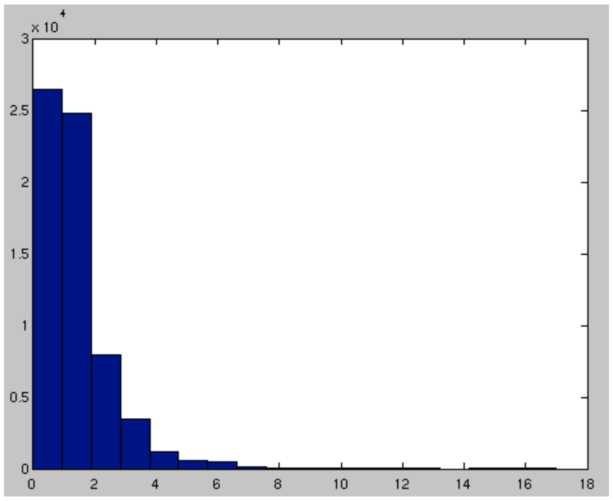
Interestingly, the number of overloaded methods has increased over time, but the proportion of overloaded methods has been reduced. Also worth noting is the number of classes with default constructors. Stylos [16] has demonstrated that programmers prefer and are more efficient when a default constructor is provided; however, not all classes have a default constructor. Instead, there was an increase in methods that returned an API-defined object (these are objects whose class is defined in the same package as the method). This may be an indication that the API is moving towards a factory pattern (or singleton pattern) for object creation. A quick overview of the documentation does indicate that this is in fact the case, although none of the methods are named with the keyword “factory”, as is the standard in Java.

Part III - The Java 5 API

An interesting analysis would have been to evaluate how the Java API evolved over time, but there were numerous issues relating to JRE compatibility when trying to load these JREs. The implementation of the evaluation tool uses generics heavily, which were only introduced in Java 5. All versions of the JRE were downloaded, and were run through the evaluation tool, but (presumably) due to deprecation and conflicts between the runtime and the tool, many classes failed to load, so the results from those analyses were difficult to trust.



Methods per class



Parameters per method

A similar analysis to the XML APIs was done on the Java 5 API. In the histogram above, we can see the same trend discovered in the XML APIs: the majority of the classes have a fewer number of methods, in this case somewhere around 30 or fewer. This lends some credit to the conclusion in part 1 that a number of methods between 20 or 30 may be a good metric for the number of methods in each class; however, Java is a huge API, with over 9000 public classes available to the programmer. An interesting observation here is that although most of the classes have fewer than 30 methods, there are numerous classes with many more methods than that. A few classes were removed from the analysis, specifically, 14 classes that had more than 150 methods each. Two classes of interest appear in this list, both of which are from the Corba logging package. One class, OMGSystemException has 526 methods, and ORBUtilSystemException has 1190 methods! OMG system exception is right.

As for the parameters per method analysis, Java ranks lower than the other APIs investigated. Remember that there some 60,000 methods in the Java API so it may be difficult to judge from the histogram, but there are many methods that require 5 or more parameters. Specifically, there are 1358 methods requiring 5 or more parameters.

Conclusions from the preliminary results

Gathering statistics about APIs seems like an interested and untapped resource when considering API usability. The issue now is: what conclusions can we draw from these metrics?

The literature is consistent in stating “less is more” [1,2,3,4,16], preferring the simpler implementations to the complicated ones. Generally, it is difficult to say what the conclusions should be, since some APIs seem better in one analysis, but then seem worse when looking at the same property from a different angle. The metrics described in this project are a first step into automatic evaluation of API usability, but barely breaks the surface of the potential. Usability is a complicated, subjective idea, thus making automated analysis difficult at best.

Future work

Throughout this report, I have discussed some of the features I wished would have been included in this project. I also mentioned some of the issues with the automatic approach of evaluating API usability. Most importantly, it is not known if using an automatic approach is really a valid measure of API usability. It may be that user studies are the best way to decide if an API is usable or not, so in the future, automatic evaluation should take place before the user study to see if the quantitative results found from the evaluation tool match the qualitative results from the user studies.

Although the validity of this approach has yet to be proven, I do believe that it is of value, especially if more subjective analysis were able to be automated. I often mentioned the semantic analysis of the naming used in the API and how this could potentially be the most interesting metric for evaluating API usability. For example, having a method *get* in one class and the method *elementAt* in another class could be considered inconsistent naming within the API, as is the case with Java's Collection classes. Also, using *length*, *size*, and *count* to mean the number of elements in some collection may seem strange to new programmers of an API (or even veteran programmers!).

Lastly, the evaluation tool can already output much more data than has been analyzed in this report. It would be interesting to take the remaining data and see if the other metrics defined in this project do in fact provide valuable insight into the usability of an API.

Conclusion

Usability is a very subjective term, and is related to both the task the user wishes to accomplish and the context in which they wish to accomplish the task. An interface's usability can be judged on a series of different measures, which differ depending on the source of the measures. Programmers are users too, but the interface they are often using are the APIs provided to them by other programmers.

The evaluation tool implemented in this project is a good first step in discovering whether automatic API usability evaluation is feasible. It collects an enormous amount of data related to a series of metrics developed from guidelines, heuristics, and qualitative studies on API usability from the literature. Three preliminary studies were conducted: one on six domain-specific APIs, another on an API's versions over time, and the third on the Java 5's JRE itself. A small portion of the data was analyzed. From the domain-specific study, many similarities were found between the different APIs, but there were also some conflicts when looking at the results from different perspectives. All together, the three studies show what type of automatic evaluation can be done on an API, but these results should be combined with user studies to see if the quantitative results from the evaluation match the subjective input of the users.

References

[1] Bloch, J., *Effective Java Programming Language Guide*, Sun Microsystems, Mountain View, CA, 2001.

[2] Bloch, J. "How to Design a Good API and Why it Matters", keynote in Library-Centric Software Design Workshop, <http://lcsd05.cs.tamu.edu/slides/keynote.pdf>, 16 October 2005

[3] Clarke, S., "Measuring API Usability", Dr. Dobbs Journal, 2004, pp. S6-S9.

[4] Clarke, S., "API Usability and the Cognitive Dimensions Framework", http://blogs.msdn.com/stevencl/archive/2003/10/08/57040.aspx 2003.

[5] Ellis, B., Stylos, J., Myers, B., "The Factory Pattern in API Design: A Usability Evaluation", IEEE ICSE, 2007, pp. 302-312.

[6] Fowler, M., "Public versus Published Interfaces", in IEEE Software, vol. 19, 2002, pp. 18-19.

[7] Hilbert, D., Redmiles, D., "Extracting Usability Information from User Interface Events", ZMC Computing Surveys, Vol. 32, No. 4, December 2000, pp. 384-421.

[8] Holmes, R., Walker, R. J., Murphy, G. C., "Strathcona example recommendation tool", ACM SIGSOFT, 2005, pp. 237-240.

[9] Johnson, R. E., "Documenting frameworks using patterns", Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 1992, pp. 63–72.

[10] McLellan, S.G., Roesler, A.W., et al, "Building More Usable APIs", Software, IEEE, 1998, 15(3) pp 78-86.

[11] Mandelin, D., Xu, L., Bodik, R., Kimelman, D., "Jungloid Mining: Helping to Navigate the API Jungle", ACM SIGPLAN, 2005, pp. 48-61.

[12] Nielsen, J., Molich, R., "Heuristic Evaluation of user interfaces", Proc. ACM CHI'90, April 1990, pp. 249-256

[13] Nielsen, J., "Usability Metrics: Tracking Interface Improvements," IEEE Software, Nov. 1996, pp. 12-13.

[14] Stylos, J., Myers, B., "Mapping the Space of API Design Decisions", IEEE Proc. VLHCC , 2007, pp. 50-60.

[15] Stylos, J., Myers, B., “The Implications of Method Placement on API Learnability”, ACM SIGSOFT, 2008, pp. 105-112.

[16] Stylos, J., Clarke, S., “Usability Implications of Requiring Parameters in Objects’ Constructors”, ACM ICSE’07, 2007, pp. 529-539.

[17] Zibran, M. F., “What Makes APIs Difficult to Use?”, IJCSNS, Vol. 8, 2008, pp. 255-261.

[18] “Using IntelliSense”, http://msdn.microsoft.com/en-us/library/hcw1s69b(VS.80).aspx 2003.

[19] ISO Standard 9241-11 (ISO 1993c), http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=16883, 1998

Acknowledgements

This project would not have come together without the help of several people. Mike Lippold, who suggested using the reflection API instead of XML parsing; Carl Gutwin and Regan Mandryk, who are always supportive and gave great ideas throughout the entire project; Paul Muir at Saint Mary’s University, who helped develop some ideas about API usability; Brian de Alwis, who has given suggestions, guidance, and a helping hand at every stage of the process in this project; and all of the members of the interaction lab, who are always around as a sounding board when ideas need to be bounced off someone. Thanks!